Cahiers GUTenberg

DOCUMENTATION DE PROJETS EN XML

Frédéric Boulanger, Yolaine Bourda

Cahiers GUTenberg, nº 35-36 (2000), p. 15-23.

<http://cahiers.gutenberg.eu.org/fitem?id=CG_2000___35-36_15_0>

© Association GUTenberg, 2000, tous droits réservés.

L'accès aux articles des *Cahiers GUTenberg* (http://cahiers.gutenberg.eu.org/), implique l'accord avec les conditions générales d'utilisation (http://cahiers.gutenberg.eu.org/legal.html). Toute utilisation commerciale ou impression systématique est constitutive d'une infraction pénale. Toute copie ou impression de ce fichier doit contenir la présente mention de copyright.

Documentation de projets en XML

Frédéric Boulanger et Yolaine Bourda

Supélec – Service Informatique
Plateau de Moulon
3 rue Joliot-Curie, F-91192 Gif-sur-Yvette cedex
{Frederic.Boulanger, Yolaine.Bourda}@supelec.fr

Résumé. Le manque de documentation de la plupart des logiciels, et notamment de ceux développés par nos étudiants, nous amène à envisager une approche de la documentation par annotation du code dans les commentaires. Nous généralisons cette approche afin de pouvoir documenter du code en n'importe quel langage, et de créer de la documentation à n'importe quel format. XML est toutefois le format que nous privilégions car ce travail s'inscrit dans un projet plus vaste de traitement des documents.

Abstract. *The lack of documentation in most software projects, and particularly in our students' projects, lead us to develop a way to document software by annotating it in comments. We generalize this approach to be able to document code in any language and to create documentation in any format. However, XML is our preferred choice since this work is part of a broader document processing project.*

Mots-clés : Documentation, annotations documentaires, programmation.

Keywords: Documentation, documentary annotations, programming.

1. Introduction

À de trop rares exceptions près, la plupart des logiciels souffrent d'un manque de documentation qui nuit à leur maintenance et à la réutilisation de leur code, quand cela ne nuit pas directement à leur développement.

Les quelques rares exceptions sont soit l'œuvre de développeurs particulièrement consciencieux, soit le résultat d'une programmation littéraire très peu répandue, bien que d'importance pour notre communauté TEX. Nous ne considérons pas le cas de JavaDoc qui, bien qu'intéressant pour documenter les services fournis par du code Java (ce que l'on appelle une API[1]), ne permet de documenter ni l'utilisation d'un programme, ni son fonctionnement interne.

1. *Application Programming Interface* ou interface de programmation d'application

La programmation littéraire est une bonne idée en ce qu'elle regroupe le code et la documentation dans le même document. C'est en effet une condition nécessaire (mais hélas pas suffisante) pour que la documentation soit à jour vis à vis du code qu'elle est censée documenter.

Malheureusement, ce style de programmation, tel qu'il existe à l'heure actuelle avec web et cweb, a quelques inconvénients qui expliquent peut-être son manque de succès. Tout d'abord, avant de pouvoir compiler un programme, il faut extraire le code du source web. Il faut pour cela disposer des outils associés au style de programmation. Pour éviter ce problème lorsqu'on distribue le code source d'un programme, on peut être tenté de distribuer le code déjà extrait du « méta-source ». Dans ce cas, on distribue un code illisible, ce qui revient à interdire à l'utilisateur potentiel de comprendre comment il fonctionne et même de l'adapter à sa plateforme ou à ses besoins.

L'approche choisie par D. E. Knuth était pertinente à l'époque à laquelle TeX a été créé, car les outils de développement n'offraient pas ce qu'apportait le système web, et les performances des ordinateurs incitaient à ne fournir à un compilateur que le strict minimum permettant d'obtenir le programme voulu. De nos jours, les langages de programmation supportent la modularité et disposent de mécanismes puissants de vérification du respect des interfaces. Les performances des machines permettent de tolérer qu'un compilateur traite plus de lignes de commentaires que de lignes de code. Une autre approche de la documentation de code est donc possible, et nous pensons qu'elle peut être utilisée pour tout type de développement, et pas uniquement pour la programmation en C, C++ ou Java.

2. Principe

Le principe de notre approche de la documentation de code est que ce code doit rester compilable directement, sans transformation préalable. Cela permet de distribuer le code source documenté sans obliger les utilisateurs à installer l'outil de documentation, tout en les faisant bénéficier de cette documentation. La conséquence immédiate de ce principe est que la documentation se trouve dans les commentaires.

Sur ce point, nous sommes en retrait par rapport à la programmation littéraire puisque nous commentons du code alors que dans un source web, on code le commentaire du programme. Cet inconvénient nous semble largement compensé par les autres avantages de notre approche.

Une autre conséquence de ce principe est que la seule chose que l'outil de documentation doit connaître du langage est la façon d'isoler les commentaires.

Cela permet donc de construire un outil capable de fonctionner avec plusieurs langages, et aussi d'utiliser plusieurs langages dans un même projet.

D'autres outils appliquent le principe d'un code source directement utilisable, mais ils ne fonctionnent que pour un langage de programmation et un format de documentation. Par exemple, `c2latex` de John D.RAMSDELL, que nous avons utilisé avant de lancer ce projet, ne fonctionne qu'avec C ou C++ pour créer de la documentation en LaTeX.

Deux problèmes subsistent : les commentaires peuvent ne pas être entièrement consacrés à la documentation, et cette documentation doit être structurée afin de pouvoir être exploitée automatiquement (documentation en ligne, environnement de programmation gérant la documentation etc). Le premier problème est important car les commentaires sont une niche syntaxique dans laquelle trouvent refuge de nombreuses espèces d'annotations : outre les commentaires bruts du programmeur, on y trouve des indications sémantiques pour divers outils de vérification ou d'aide à la programmation, et on risque maintenant d'y trouver de la documentation — un comble ! La documentation sera donc isolée dans les commentaires entre deux chaînes de caractères définissables par l'utilisateur, et que nous représentons ici par les pseudo-balises `<xdoc>` et `</xdoc>`.

Cette façon d'exploiter des structures particulières dans les commentaires d'un langage est assez classique, et nous incite à considérer cette approche de la documentation de code comme de la « documentation par annotation » plutôt que comme de la programmation littéraire. C'est en effet la documentation qui suit le fil du code et non le contraire. Si cette façon de présenter les choses ne convient pas, il est toujours possible d'exploiter ensuite la structure de la documentation, afin de changer l'ordre dans lequel les différentes parties sont présentées.

Pour structurer la documentation, LaTeX aurait pu convenir, mais il est paradoxalement trop puissant pour ne décrire qu'une structure logique. En effet, même si LaTeX donne de l'importance à la structure logique d'un document et tente d'isoler les problèmes de mise en forme dans des classes de documents et des extensions, il s'agit quand même d'un langage qui permet de faire de la mise en forme. De plus, la structure lexicale de TeX rend difficile l'analyse d'un document, tout simplement parce que certaines « instructions » ont le pouvoir de modifier les règles de grammaire — le fait que les programmes autres que TeX qui tentent de lire du LaTeX s'étranglent au premier changement de `\catcode` non prévu illustre le problème...

À l'autre extrême, HTML est trop pauvre car il ne permet pas de choisir la façon de structurer la documentation. De plus, il comporte à la fois des balises exprimant une structure logique (par exemple `<LI>` pour élément d'une liste), et des balises exprimant un aspect visuel (`<B>` pour passer en gras). Il nous faut

donc un langage de balisage logique qui ne permette pas d'exprimer des indications de mise en forme, et qui nous permette de structurer la documentation à notre guise. Il s'agit bien sûr d'XML.

3. Détails pratiques

Lorsqu'on documente du code, il peut être intéressant d'en inclure une portion dans la documentation. Dupliquer le code dans la documentation n'est pas envisageable car la redondance entraîne tôt ou tard des oublis de mise-à-jour. Il faut donc intégrer à la documentation le véritable code source et non une copie de ce code.

Cela signifie que tout ce qui apparaîtra dans la documentation n'est pas forcément dans les commentaires. Autrement dit, les balises `<xdoc>` et `</xdoc>` ne sont pas forcément équilibrées dans un commentaire. Lorsqu'on sort d'un commentaire à l'intérieur d'un bloc de documentation, le lexème qui fait sortir du commentaire est ignoré, de même que le lexème qui fait entrer dans le commentaire contenant le `</xdoc>` correspondant. Il est ainsi possible de faire apparaître des commentaires dans le code intégré à la documentation.

Par exemple, le fragment de code C suivant :

```
/* <xdoc>La nature des tableaux en C permet d'écrire
   des choses comme : <code language="C"> */
assert(tab[i] == i[tab]); /* c'est vrai ! */
/* </code> qui sont vraies bien que troublantes.
   </xdoc> */
```

donne le code XML :

```
  La nature des tableaux en C permet d'&eacute;crire
  des choses comme : <code language="C">
  assert(tab[i] == i[tab]); /* c'est vrai ! */
  </code> qui sont vraies bien que troublantes.
```

On remarque que la fermeture du premier commentaire et l'ouverture du troisième commentaire ont disparu lors de l'extraction du code XML. Par contre, le second commentaire est conservé tel quel. On remarque aussi que le 'é' de 'écrire' a été transformé en `é` pour respecter le codage ASCII standard du fichier XML.

Ce fragment de code XML pourrait être rendu comme suit sur papier :

La nature des tableaux en C permet d'écrire des choses comme :

```
assert(tab[i] == i[tab]); /* c'est vrai ! */
```

qui sont vraies bien que troublantes.

La documentation complète est composée de la documentation extraite du code, mais aussi de fichiers de documentation « pure » qui pourront faire référence aux annotations du code source.

4. Quelle(s) DTD ?

Une fois construite, la documentation doit être exploitée de diverses façons : manuel d'utilisation, référence pour développeurs, description des algorithmes utilisés etc. Chacune de ces « vues » de la documentation peut utiliser différents média : aide en ligne, documentation papier, documentation interactive dans un environnement de développement...

L'outil d'extraction de documentation n'impose aucune contrainte sur la DTD, chacun peut donc utiliser la plus adaptée à chacun de ses projets. Toutefois, le développement d'outils ou de feuilles de style XSL permettant d'exploiter cette documentation brute bénéficierait de l'existence d'une DTD de documentation suffisamment générale et souple pour s'adapter à différents types de projets.

Il semble toutefois probable qu'une DTD unique ne pourra pas satisfaire tous les besoins de documentation, à moins de devenir tellement lâche qu'elle ne définisse même plus un véritable type de document. Une autre approche consiste à définir une DTD paramétrable, un peu de la même façon qu'une classe de documents LaTeX peut être utilisée avec différentes options.

La DTD de base, assez pauvre, code uniquement la structure minimale pour qu'une documentation puisse être traitée par des outils génériques — archivage, indexation... Pour un type de documentation particulier, on utilise des extensions de la DTD de base afin de construire une DTD sur mesure.

Les domaines nominaux d'XML permettent d'utiliser des éléments de DTDs d'extensions dans un document respectant une DTD de base (on peut ainsi coder des équations à l'aide de MathML dans un rapport technique). Cela conduit toutefois à un document qui n'est valide ni par rapport à la DTD de base, ni par rapport aux DTD d'extensions. De plus, dans notre cas, la syntaxe de la documentation serait alourdie car il faudrait préfixer chaque élément du nom du domaine associé à l'extension dans laquelle on souhaite l'interpréter.

Une solution est de générer automatiquement la DTD d'un document à partir de sa DTD de base et des extensions utilisées. Nous étudions cette possibilité qui permettrait d'alléger la syntaxe de la documentation.

Cet aspect est important car, même si les outils de l'environnement de développement permettent de saisir la documentation de façon naturelle, son codage dans les commentaires doit rester lisible. Il faut donc absolument éviter que les balises et les attributs prennent plus de place que la documentation.

5. Généralisation

Extraire de la documentation revient finalement à déterminer ce qui est de la documentation dans le fichier source, et à l'écrire dans un fichier destination. Ceci nous amène à enrichir les marqueurs de documentation afin de pouvoir préciser dans quel fichier la documentation doit être placée. De plus, rien n'empêche de traiter autre chose que de la documentation au sens classique du terme puisque notre outil n'interprète pas ce qu'il extrait.

La documentation d'un projet peut parfois être exprimée de manière formelle, c'est-à-dire exploitable systématiquement. L'intérêt d'une documentation formelle est qu'elle ne peut être interprétée que d'une seule façon, et que sa validité peut être vérifiée automatiquement.

Par exemple, dans un programme écrit en langage C et destiné à contrôler un composant programmable, le programmeur fait des hypothèses sur le fonctionnement du composant. Ces hypothèses doivent être documentées afin qu'il soit possible de vérifier qu'elle sont en accord avec les spécifications du composant.

Dans le cas d'un composant programmable, ces hypothèses peuvent prendre la forme d'un fichier de tests que l'outil de programmation pourra faire passer au composant. On voit donc que la documentation d'un code source peut prendre la forme d'un autre code source !

L'exemple suivant montre comment le contenu du fichier de test peut être inclus dans la documentation. Pour cela, les pseudo-balises `xdoc` et `/xdoc` ont un attribut `out` qui permet de préciser que leur contenu doit être placé dans un fichier particulier :

```
/* <xdoc><prerequis>
   On suppose ici que les sorties <var>s1</var>
   et <var>s2</var> du composant ne sont jamais
   actives ou inactives en même temps.
   Le fichier <fichier href="./compoZZZ.tst"/>
   contient un test vérifiant ce prérequis.
   </prerequis>
   </xdoc>
   <xdoc out="./compoZZZ.tst">
   s1 .xor. s2 = true;
   </xdoc>
*/
void uneFonction() {
  ...
}
```

Notre outil d'extraction peut donc être considéré comme un programme qui repère des portions balisées dans un fichier source et les écrit dans d'autres fichiers. Mais nous allons voir que « lire » ou « écrire » un fichier exige certaines précautions.

6. Espaces d'entrée et de sortie

Nous avons déjà vu que lorsqu'on écrit dans un fichier XML, certains caractères littéraux doivent être traduits : un `&` doit être écrit `&`. Si on écrivait dans un fichier LaTeX, le même `&` devrait être écrit `\&`.

La nécessité de coder certains caractères de façon particulière peut être représentée par la notion d'espace de sortie. Ainsi, dans l'espace de sortie XML, l'esperluette se code `&` et le signe « inférieur strict » `<`. Il faut bien noter qu'il s'agit de l'esperluette et du signe « inférieur strict » en tant que symboles, pas en tant que caractères. En effet, une balise XML débute par le caractère `<`, qui ne doit pas être remplacé par `<`. De même, le caractère `&` délimite à gauche les entités XML, et lorsqu'il est utilisé pour cela, il ne doit pas être remplacé par `&`.

La notion d'espace de sortie permet à notre outil d'extraire de la documentation sous d'autres formats qu'XML — il suffirait de coder l'espace de sortie LaTeX dans notre outil pour qu'il permette d'écrire des documents « normaux » — et de créer des fichiers en divers langages. Mais cette notion n'est pas suffisante : il est aussi nécessaire d'indiquer comment interpréter les caractères du fichier source.

Nous avons vu dans un exemple que le caractère ‘é’ dans le fichier source était traduit en `é` dans l’espace de sortie XML. Cette façon de dire les choses est partiellement fausse : c’est en effet le symbole « e accent aigu » qui est codé ainsi en XML. Il manque ce qui a permis à l’outil de reconnaître un « e accent aigu » dans le caractère ‘é’ du fichier source. Ce qu’on a lu n’est qu’un nombre, et il faut connaître le codage du document pour l’interpréter comme un ‘é’. Toutefois, l’interprétation du contenu du fichier source ne se limite pas à l’utilisation d’un codage des caractères.

Reprenons notre exemple du fichier de tests pour un composant programmable, et supposons que dans le langage utilisé pour exprimer ces tests, le OU exclusif se note `*/`. Le code de ces tests fait partie de la documentation du code C chargé de communiquer avec ce composant ; il se trouve donc dans un commentaire du langage C. En C, les commentaires sont délimités par les lexèmes `/*` et `*/`, ce qui fait que l’apparition d’un OU exclusif de notre langage de test dans un commentaire a pour effet de clore le commentaire...

Être dans un commentaire du langage C est donc une condition qui nous interdit d’utiliser certains caractères. Nous devons donc coder le OU exclusif de nos tests à l’aide de caractères autorisés, mais dans le fichier de tests, ce OU exclusif doit apparaître sous la forme `*/`. De la même façon qu’un espace de sortie indique comment coder certains symboles, un espace d’entrée indique comment interpréter certains caractères. Dans notre cas, on pourra décider de coder le OU exclusif sous la forme `.xor.` par exemple, et construire un espace d’entrée dans lequel `.xor.` s’interprète en `*/`.

7. Conclusion

L’approche de la documentation que nous proposons tente de conserver certains bons aspects de la programmation littéraire tout en s’appuyant plus fortement sur les mécanismes fournis par les langages de programmation actuels. Elle permet la distribution du code source documenté en éliminant l’étape d’extraction du code compilable, ce qui lui permet de plus de s’appliquer à tout langage de programmation dont on sait isoler les commentaires, et d’utiliser plusieurs langages dans un même projet.

En choisissant XML pour structurer la documentation, nous souhaitons permettre à des outils de projeter différentes « vues » de cette documentation qui pourront s’adresser à différents publics : utilisateurs, programmeurs, techniciens de maintenance. Chacune de ces vues pourra de plus être présentée sur différents média : manuel papier, aide en ligne, système de maintenance assistée par ordinateur.

La mise au point d'un système de DTDs paramétrables permettra de structurer la documentation de la manière la plus appropriée, tout en bénéficiant d'outils capables de la traiter selon une DTD de base.

Le support de plusieurs fichiers de documentation et les notions d'espace d'entrée et de sortie permettent de traiter correctement des portions de documentation exprimées dans un langage formel, et ce quelque soit le langage hôte de la documentation.

Cet outil est actuellement en version de développement et devrait être mis à la disposition d'étudiants n'ayant pas participé à son développement dès la rentrée prochaine. Nous pourrons alors juger de son impact sur la qualité de la documentation des projets...